METHOD AND APPARATUS FOR RESTORING REGISTERS AFTER CANCELLING A MULTI-CYCLE INSTRUCTION

BACKGROUND

[0001] A programmable processor, such as a microprocessor for a computer or a digital signal processing system, may support one or more "multi-cycle" machine instructions in which a single machine instruction directs the processor to perform multiple operations. An exemplary multi-cycle instruction is a "Load Multiple" instruction in which the processor performs a series of load operations in response to a single machine instruction. Another example is a "Push-Pop Multiple" instruction that directs the processor to push or pop multiple registers to or from a stack. Because multi-cycle instructions pack multiple operations into a single machine instruction, they may increase code density and improve the operational efficiency of the programmable processor.

DESCRIPTION OF DRAWINGS

[0002] Figure 1 is a block diagram illustrating a pipelined programmable processor according to an embodiment.

[0003] Figure 2 is a schematic illustrating an exemplary execution pipeline.

[0004] Figure 3 is a schematic illustration of a portion of a pipeline, which includes a speculative commit register according to an embodiment.

[0005] Figure 4 is a flowchart describing a speculative commit operation according to an embodiment.

[0006] Figure 5 is a block diagram of a mobile video unit including a processor according to an embodiment.

DESCRIPTION

[0007] Figure 1 is a block diagram illustrating a programmable processor 100, which supports one or more multi-cycle instructions (MCIs). The processor 100 may include an execution pipeline 102 and a control unit 104. The control unit 104 may control the flow of instructions and data through the pipeline 102 in accordance with a system clock 105. During the processing of an instruction, the control unit 104 may direct the various components of the pipeline to decode the instruction and to perform the corresponding operation including, for example, writing results back to memory.

[0008] Instructions may be loaded into a first stage of the pipeline 102 and be processed through subsequent stages. A stage may process concurrently with the other stages. Data may be passed between the stages in the pipeline 102 in accordance with the system clock signal. Instruction results may emerge at the end of the pipeline 102 in succession.

[0009] In response to an MCI, a stall controller 106 may stall one or more stages of the pipeline 102 by asserting stall signals 108 in order to prevent the pipeline 102 from fetching and decoding additional instructions while the MCI is executing. After stalling a portion of the pipeline 102, an MCI controller 110 may assert MCI signals 112 and direct the pipeline 102 to perform additional operations defined by the current MCI.

[0010] Figure 2 illustrates an exemplary pipeline 102. The pipeline 102 may include, for example, five stages: instruction fetch (IF), instruction decode (DEC), address calculation (AC), execute (EX), and write back (WB). According to alternate embodiments, these stages may include sub-stages, e.g., the EX stage may include multiple sub-stages EX1, EX2, etc.

[0011] Instructions may be fetched from a memory device such as, for example, a main memory or an instruction cache, during the IF stage by a fetch unit 200 in a clock cycle. An instruction fetched in a clock cycle may be decoded in a subsequent clock cycle in the DEC stage by an instruction decode unit 202. The results may be passed to the AC stage, where a data address generator (DAG) 204 may calculate memory addresses for performing the operation. During the EX stage, an execution unit 206 may perform a specified operation such as, for example, adding or multiplying two numbers. The execution unit 206 may contain specialized hardware for performing the operations

including, for example, one or more arithmetic logic units (ALUs), multiply and accumulate (MAC) units, and barrel shifters. A variety of data may be applied to the execution unit 206 such as the addresses generated by the DAGs 204, data retrieved from memory or data retrieved from data registers 208. During the WB stage, the results may be written back to data memory or to data registers 208.

[0012] During execution of an MCI, multiple instructions may be issued from the DEC stage of the pipeline 102 over several clock cycles. The MCI remains stalled in the decode stage of the pipeline 102 while multiple "sub-instructions" may be sent down the pipeline 102 under control of the MCI controller 110. The MCI controller 110 may operate according to a number of internal state machines in order to direct the instruction decode unit 202 to dispatch a number of operations over a number of clock cycles during the execution of the MCI.

[0013] The stall controller 106 may stall one or more stages of the pipeline 102 by asserting stall signals 108 in order to prevent the pipeline 102 from fetching and decoding additional instructions while the MCI is executing. The stages of the pipeline 102 may include storage units, such as stage registers 210, for storing the results of the current stage. The stage registers 210 may latch the results according to the system clock. The stall signals 108 received by the stage registers

210 control whether or not the stage registers 210 latch the results from the previous stage. In this manner, the stall controller 106 may stall one or more stages of the pipeline 102 in response to an MCI.

[0014] An MCI may include a terminal sub-instruction, i.e., the last sub-instruction, and one or more non-terminal sub-instructions, which include the first and any intervening sub-instructions. When a sub-instruction reaches the WB stage, that sub-instruction is considered to be committed. When the terminal sub-instruction reaches the WB stage, the MCI is considered to be committed.

[0015] An instruction may be cancelled (i.e., "killed"), and all writes turned off for that instruction, if it is no longer valid for the current program flow. This may occur, for example, when an interrupt is taken. When an interrupt occurs, all instructions in the pipeline may be cancelled, e.g., by placing zeroes into the pipeline latches of the cancelled instructions, and instructions from an interrupt service routine (ISR) may be fetched and introduced into the pipeline.

[0016] After the interrupt has been handled by the ISR, the program counter (PC), which tracks the program flow, may return to a cancelled instruction to resume the program flow. In other words, the pipeline backs up to the state it had prior to executing the cancelled instruction.

[0017] When the PC returns from the ISR, it may be desirable for the architectural registers to have the values they had before the cancelled instruction was introduced into the pipeline. The architectural registers may include, for example, pointer registers (PREG) for storing pointer values.

[0018] When an MCI is cancelled in the pipeline, a non-terminal sub-instruction may have already reached the WB stage, and in doing so, may have written a result to an architectural register. The previous value held by that architectural register may be lost, making it difficult for the processor 100 to return to the state it had prior to executing the cancelled MCI.

[0019] In an embodiment, results generated during the execution of non-terminal sub-instructions of an MCI in the pipeline 102 may not be written to the architectural registers until the MCI commits, which occurs when the terminal sub-instruction reaches the WB stage. Figure 3 illustrates an exemplary pipeline 300 that includes a speculative commit register (SCR) 302 to store a value produced by a non-terminal sub-instruction until the clock cycle in which the MCI commits. When a non-terminal instruction reaches the WB stage, any results may be written to the SCR 302 rather than an architectural register 304. When the terminal sub-instruction reaches the WB stage, the MCI controller 110 may control a

multiplexer (MUX) 306 to write the value stored in the SCR 302 to the architectural register 304. In this manner, architectural registers are not written until the MCI commits. Thus, if the MCI is cancelled in the pipeline 300 prior to committing, the pipeline may be restored to the state it had prior to the MCI being executed.

[0020] Figure 4 is a flowchart illustrating a speculative commit operation 400 according to an embodiment. The flow of the operation described in Figure 4 is exemplary, and blocks in the flowchart may be skipped or performed in different order according to alternate embodiments.

[0021] When a sub-instruction reaches the WB stage in block 402, the processor 100 determines whether or not the sub-instruction is the terminal sub-instruction in block 404. For a non-terminal sub-instruction, it is determined whether the result is designated for an architectural register in block 406. If so, the result is written to the SCR 302. If it is determined that the MCI is cancelled in the next cycle in block 408, the operation 400 ends with the architectural register unaltered by the cancelled MCI. If the MCI is not cancelled, the operation 400 returns to block 402.

[0022] When the terminal sub-instruction for the MCI reaches the WB stage, the operation corresponding to that sub-instruction is performed in block 412, which may include writing

to an architectural register. If it is determined that a result from a non-terminal sub-instruction of the MCI was written to the SCR 302, the MCI controller 110 controls the MUX 306 to write that result to the corresponding architectural register 304. Otherwise, the operation 400 ends.

[0023] A "Link" instruction is an example of an MCI that may alter an architectural register before committing. The Link instruction may be used to invoke a subroutine. When a subroutine is called, the processor 100 may store a return address for the subroutine on a stack, and set aside space on the stack (a frame) to store dynamic local data for the subroutine during its execution.

[0024] The stack pointer points to the top of a stack, and changes often during the execution of a program. The size of the stack is increased on each subroutine call by decrementing the stack pointer, which grows downwards. Later, on subroutine return, the size of the stack may be decreased by incrementing the stack pointer appropriately.

[0025] When the subroutine is called, the frame pointer may be set to the value the stack pointer had when the current subroutine was called, before it was decremented for the subroutine. Because the stack pointer may change during execution, the data stored on the stack for the current subroutine are typically de-referenced by the frame pointer,

since the frame pointer stays constant during the execution of the subroutine.

[0026] An exemplary Link instruction includes the following four sub-instructions:

1 PUSH RETS

2 PUSH FP

3 FP = SP

4 SP = SP + IMM

[0027] These sub-instructions may result in the processor 100 (1) pushing a return address for a subroutine (RETS) on a stack, (2) pushing a frame pointer (FP) on the stack, (3) moving the stack pointer (SP) to the frame pointer, and updating the stack pointer based on a frame size (IMM for immediate value) for the subroutine as specified by the instruction. Typically, when sub-instruction (3), FP = SP, reaches the WB stage, the stack pointer value is written to an architectural register reserved for the frame pointer, FPREG. Since this occurs before the MCI commits, the previous value in FPREG would be lost if the Link instruction was cancelled before terminal sub-instruction (4) reached the WB stage.

[0028] According to an embodiment, this problem may be avoided by storing the stack pointer value in the SCR 302 until the Link instruction commits. Referring now to Figure 4, when instruction (3) reaches the WB stage in block 402, it is

determined to be a non-terminal sub-instruction in block 404. Since the result is designated for the FPREG, the result, SP, is written to the SCR 302 rather than FPREG. When terminal sub-instruction (4) reaches the WB stage in block 402, the stack pointer is updated and written to an architectural register reserved for the stack pointer, SPREG, in block 412, and the stack pointer value in the SCR 302 is written to FPREG in block 414.

[0029] An "Unlink" instruction is another example of an MCI that may alter an architectural register before committing. The Unlink instruction may be used to exit a subroutine. An exemplary Unlink instruction includes the following three sub-instructions:

1 RETS = [FP + 4]

2 SP = FP + 8

3 FP = [FP]

[0030] These sub-instructions may result in the processor 100 (1) restoring the return address from the stack, (2) restoring the stack pointer, and (3) restoring the frame pointer with a value read from memory. Typically, the architectural register SPREG would be written to when sub-instruction (2), SP = FP + 8, reached the WB stage. Since this occurs before the MCI commits, the previous value in SPREG would be lost if the Unlink

instruction was cancelled before terminal sub-instruction (3) reached the WB stage.

[0031] According to an embodiment, this problem may be reduced by storing the updated stack pointer value in the SCR 302 until the Unlink instruction commits. As shown in Figure 4, when instruction (2) reaches the WB stage in block 402, it is determined to be a non-terminal sub-instruction in block 404. Since the result is designated for the SPREG, the result of FP + 8 is written to an SCR rather than SPREG. When terminal sub-instruction (3) reaches the WB stage in block 402, the frame pointer is restored in block 412, and the value in the SCR 302 is written to SPREG in block 416.

[0032] A "PushPopMultiple" instruction is another example of an MCI that may alter an architectural register (SPREG) before committing. The PushPopMultiple instruction may be used to perform a number of pushes or pops from the stack in sequence. As each sub-instruction exits the AC stage, the SP value is incremented, or decremented, by a value of one. An SP value calculated in the AC stage in response to a sub-instruction may be forwarded to a working register, or future file (FF) 310, in the DEC stage. This new SP value may be used as the base SP value for the address calculation operation performed in response to the next issued sub-instruction. The changing SP values may be stored in the SCR 302 until the terminal sub-

instruction reaches the WB stage, at which point the final SP value may be written to the architectural register, SPREG.

[0033] The processor 100 may be implemented in a variety of systems including general purpose computing systems, digital processing systems, laptop computers, personal digital assistants (PDAs) and cellular phones. In such a system, the processor may be coupled to a memory device, such as a Flash memory device or a static random access memory (SRAM), which stores an operating system or other software applications.

[0034] Such a processor 100 may be used in video camcorders, teleconferencing, PC video cards, and High-Definition Television (HDTV). In addition, the processor 100 may be used in connection with other technologies utilizing digital signal processing such as voice processing used in mobile telephony, speech recognition, and other applications.

[0035] For example, Figure 5 illustrates a mobile video device 500 including a processor 100 according to an embodiment. The mobile video device 500 may be a hand-held device which displays video images produced from an encoded video signal received from an antenna 502 or a digital video storage medium 504, e.g., a digital video disc (DVD) or a memory card. The processor 100 may communicate with a cache memory 506, which may store instructions and data for the processor operations, and other devices, for example, an SRAM 508.

[0036] The processor 100 may be a microprocessor, a digital signal processor (DSP), a microprocessor controlling a slave DSP, or a processor with a hybrid microprocessor/DSP architecture. The processor 100 may perform various operations on the encoded video signal, including, for example, analog-to-digital conversion, demodulation, filtering, data recovery, and decoding. The processor 100 may decode the compressed digital video signal according to one of various digital video compression standards such as the MPEG-family of standards and the H.263 standard. The decoded video signal may then be input to a display driver 510 to produce the video image on a display 512.

[0037] A number of embodiments of the invention have been described. Nevertheless, it will be understood that various modifications may be made without departing from the spirit and scope of the invention. For example, other SCRs may be provided in the pipeline for different MCIs and different architectural registers. Accordingly, other embodiments are within the scope of the following claims.